Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 2009 Proceedings

Americas Conference on Information Systems (AMCIS)

2009

FAST ACCESS: A system architecture for RESTful Business Data

Sebastian Kloeckner
University of Augsburg, sebastian.kloeckner@wiwi.uni-agusburf.de

FAST ACCESS:
A system architecture for RESTful Business Data

Sebastian Kloeckner
University of Augsburg
sebastian.kloeckner@wiwi.uni-augsburg.de

ABSTRACT

Enterprise systems and architectures are a field of significant research, especially in regard to service-oriented architectures and web services. Web services propose great benefits and are often suggested as the new paradigm for building distributed applications and systems, but they are becoming more and more complex and consequently less understandable for humans. The web development community on the other side widely backs the resource-oriented paradigm of REpresentational State Transfer (REST). While REST has been one of the success factors of the World Wide Web, it has not been taken into deeper consideration for enterprise architectures yet.

This paper presents a new system architecture which is based on the primitives of REST and a layered architectural concept, which offers a unified interface, integrates type definitions of available resources, allows all CRUD-operations on secured disparate sources and could be and has been used for enterprise system integration purposes.

Keywords

REST, architecture, web services, service orientation, architectural layers, SOA, business data

INTRODUCTION

Service-oriented architectures (SOA) and especially their implementation with web services have gained broad attention by science and practice as they deliver extended interoperability and the possibility of service composition (Weerawarana, Curbera, Leymann, Storey and Ferguson, 2005) and they have been widely accepted in the research community as a new enterprise systems architecture paradigm (Papazoglou, Traverso, Dustdar, Leymann and Krämer, 2006). Some authors even went as far as to state: “By 2008, SOA will be a prevailing software engineering practice, ending the 40-year domination of monolithic software architecture (0.7 probability).” (Natis, 2003)

But the increasing complexity of web services, caused by the requirements of traditional enterprise computing and its expectations to quality of service (Weerawarana et al., 2005), have been object of criticism, especially from web development community (e.g. (Bosworth, 2004)), which favour a data centric composition model instead of the behavioural aggregation of services. Additionally, the architectural concept of REpresentational State Transfer (REST) had a massive impact on the way information is exchanged worldwide and the Web 2.0 has gained enormous popularity due to its simple structure of use. At the same time several integration projects at the university and in business contexts have shown that access to data of formerly unintegrated systems, even for atomic transactions only, solves a considerable number of integration issues.

But the regularly cited technologies, architectures and applications of Web 2.0, e.g. YouTube, Flickr, etc., usually use the standard functionality offered by HTTP. While adequate for these purposes, especially the type definition of transferred resource representation, which is based on the standardized MIME-definition, is inadequate for business purposes and cannot be easily extended. Therefore it seems to be advisable to develop a new system architecture, which follows the basic principles of REST and uses the concepts of HTTP, but solves the existing downfalls in order to be usable for business purposes.

This paper follows the design science approach of Hevner, March, Park and Ram (2004) and presents a system architecture, which was incrementally refined and tested during various integration projects. It is structured as follows: Section 2 presents related work, the research methodology and a short overview over the basic concepts of REST. Section 3 then illustrates the functional and structural concept of the proposed systems architecture. Section 4 provides conclusions and areas of further research.

RELATED WORK

Service-oriented architectures and web services have experienced high interest of industry and science during the last decade. Using standardized technologies and protocols (UDDI, WSDL, SOAP, WS-BPEL, etc.) (Weerawarana et al., 2005), they are often seen as the new paradigm for building distributed applications and systems. But on the other side, there is also an ongoing debate about the problems of web services. Mainly two arguments are part of the criticism: Web Services and the related technologies are perceived as being very complex (Bosworth, 2004) and are said to disagree with the basic architectural principles of the web (Rosenberg, Curbera, Duftler and Khalaf, 2008).

The resource-oriented architecture of the web, proposed by the REpresentational State Transfer (REST) model and characterized by its imposed constraints (Fielding, 2000), has proven its effectiveness, as the Web works well (Vinoski, 2007). In the recent past the mashup concept, often based on the HTTP-implementation of REST and focused on information sharing and aggregation to support content publishing for a new generation of Web applications, has emerged. "Mashups" have become one of the hottest buzzwords in the Web applications area (Parameswaran and Whinston, 2007), and many companies and institutions are rushing to provide solutions (Yu, Benatallah, Casati and Daniel, 2008). Although many have adopted the (service) mashup concept and recognized its value, realizing the concept is still challenging, and much work remains before mashup applications in a mature stage will be seen (Benslimane, Dustdar and Sheth, 2008).

While there are many successful mashup applications, e.g. Yahoo Pipes, most of them have a read-only interface and are mainly about sharing and aggregation from disparate sources (Yu et al., 2008). A comprehensive overview of the different mashup and social computing applications is given by Parameswaran and Whinston (2007). Newer approaches, like BITE (Curbera, Duftler, Khalaf and Lovell, 2007; Rosenberg et al., 2008), begin to integrate workflow concepts and present a model for integrating REST principles into a simplified workflow language. But as these concepts still use the HTTP methods PUT and DELETE, which are usually not supported by modern browsers anymore, they are only applicable for machine-to-machine communication. In addition, these concepts are mostly based on the available HTTP MIME-types and typically use the generic type "text/xml" whenever "raw data", as often the case with business data, is present. A dedicated description of transferred data types is at least not mentioned, but would be very helpful in a business environment. Furthermore, the internal structure of the proposed concepts is often not described.

To present state, at least to the knowledge of the authors, no layered architectural concept exists, which offers a unified interface, integrates type definitions of available resources, allows all CRUD-operations on disparate sources and could be and is used for enterprise application integration.

RESEARCH APPROACH

In this contribution an architectural system concept for integration of disparate business data is presented. The research approach follows the design science method of Hevner, et. al. (2004). Vaishnavi and Kuechler (2004) define the awareness of problem, suggestion, development, evaluation and conclusion as primary process steps and proposal, tentative design, artefact, performance measures and results as respective outputs of the steps of a design science approach. As shown in the introduction and related work, the web service concept has been criticized due to its complexity and its conflict with the basic architectural principles of the web. REST and mashups on the other side are intensively discussed as an additional concept for integration of distributed hypermedia systems. Therefore, this paper follows these concepts and proposes a system architecture, which further enhances the integration efforts for distributed business data. The presented concept has been incrementally tested, refined and validated during various integration projects and has shown applicability in many different data integration scenarios.

OVERVIEW OF REST

The architectural style of REpresentational State Transfer (REST) has been very successful for distributed hypermedia systems in the last years. The most famous example applying the REST architectural style is probably the Hypertext Transfer Protocol (HTTP) (Berners-Lee, Fielding and Frystyk, 1996; Fielding, Gettys, Mogul, Frystyk, Masinter, Leach and Berners-Lee, 1999). According to Fielding (2000), REST is characterized by the following properties:

- Client-Server architectural style
- Stateless communication
- Cache
- Uniform Interface
- Layered System
- Code on Demand

The Client-Server architectural style allows a separation of concern and therefore the involved components can evolve independently. The stateless communication induces better visibility (a single request datum discloses the full nature of the request), reliability (the recovering from partial failures is facilitated) and scalability (a server does not have to manage resources across requests). Caching improves network efficiency as some interactions can be completely eliminated. The uniform interface is the central feature, which distinguishes REST from other network-based architectural styles. According to Fielding (2000), "REST is defined by four interface constraints:

- identification of resources
- manipulation of resources through representation
- self-descriptive messages
- and, hypermedia as the engine of application state"

The layered system constrains the component behaviour in a way that each component cannot "see" beyond the immediate layer with which they are interacting and therefore the complexity of the overall system is limited. Finally, the code on demand style, an optional constraint of REST, reduces the number of features required to be pre-implemented by the client.

FROM REST TO FAST ACCESS

The properties of the FAST ACCESS system architecture will be shown in the following section. The presentation is based on the three different perspectives of the general systems theory of Ropohl (Ropohl, 1999). These perspectives include the functional, the structural and the hierarchical view. The functional view presents a system as a "black box" and is characterized by certain properties, which can be observed from outside. The structural view shows a system as a whole of interlinked elements. It illustrates the relationship between the elements of the system. And finally the hierarchical view emphasizes the fact that elements of a system can be interpreted as systems themselves and that they might be part of another super-system. For the presented system architecture the hierarchical view is omitted, as it is obvious due to its recursive structure.

FUNCTIONAL BEHAVIOR OF THE EXTERNAL INTERFACE (FUNCTIONAL VIEW)

REST and the HTTP with its basic methods, GET, for requesting a resource, POST, for setting of resource, PUT, for updating a resource, and DELETE, for erasing a resource, have proven to work well. From an external perspective, especially the unique interface and the unique identification by URIs are less complex when compared with web services. While these methods are adequate for machine-to- machine communication, they cannot be used for direct human-to-machine interaction, in the sense of regular web users, as most of the prevailing browsers do not support the PUT and DELETE method anymore, at least not in a simple way. The standard procedure for creating or manipulating data is usually the use of an HTML-Form in combination with a HTTP-POST. In consequence, the basic methods of HTTP cannot be used directly in human-machine-interactions (Webuser to Webserver).

Instead the usage of four resources, which emulate and enhance the basic methods of HTTP and have proven to be needed and useful in several integration projects, in combination with HTTP GET and POST are proposed: GET, POST, LIST and SEARCH. These resources are server-side processing components (server-side scripts), which offer the necessary operations and can be used in machine-to-machine as well as, in combination with XSLT, human-to-machine communications. This unified interface in combination with possible machine-to-machine communication allows a recursive interconnection with other frameworks of this type, as shown in Figure 1.

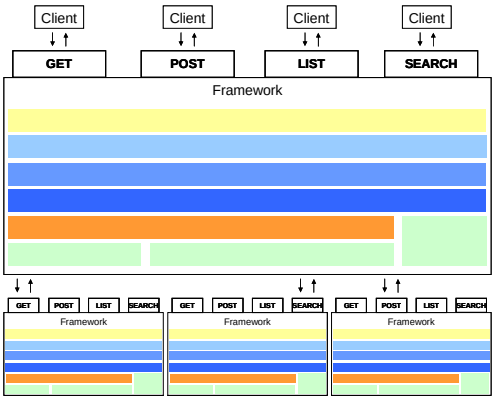

Figure 1: Generic interface of FAST ACCESS

Additionally, HTTP has one advantage, which is often overseen by other REST-based system concepts, although it allows clients type-specific processing: HTTP contains, within others, the Content-Type, based on standardized MIME-types, as header attribute for the description of the requested or delivered data. But for the exchange of business data several, sometimes conflicting, standards exist (RosettaNet, ebXML, UN/EDIFACT, SWIFT). In consequence, a system, which works with business data types, has to somehow disclose its data type structure on which the transferred data instances are based. Therefore, types and instances of these types have both to be "requestable" in the same way as representations are addressed by HTTP. This is achieved by Uniform Resource Identifiers (URI), which are realized through Globally Unique Identifiers (GUID) for type-IDs and instance-IDs. For example, requesting type-IDs results in the schema of the requested data type.

When combining the different required methods and parameters, the following function matrix is created:

FUNCTION		Headerinformation		
		None	Type-ID	Instance-ID
Body Information	GET	Find general Type Information	Find defined Type Data	Find defined Instance Data
	POST (Data)	Create Type	Update Type / Create Instance	Update Instance
	POST (No Data)	INVALID	Delete Type	Delete Instance
	LIST	Find all Types	Find all Instances	Nothing (Redirect to GET)
	SEARCH (Get)	Find Basic Types	Find type definition	Nothing (Redirect to GET)
	SEARCH (Post)	Find fitting Types	Find fitting Instance	INVALID

Figure 2: Function Matrix based on required methods and attributes

The matrix above presents the required functions, which have to be available in order to enable a simple method for exchanging data. The characteristics of each function (method/parameter combination) will be explained in the following sections. As username and password of the respective clients are an obligatory part of the QueryString for each request, they are not mentioned again in the explanation of the behaviour of the respective resources.

Behaviour of the GET-Resource

The GET-Operation (GET-resource is the target of the request) can only be called with an HTTP-GET and accordingly only QueryString-Data is allowed. A HTTP-POST with data, while possible, is ignored in order to keep the principles of REST. A request to the GET-resource without any attributes does not indicate which type or instance is requested. Accordingly this operation can be used for information about type creation. Consequently, as response to a request of this kind, the general structure of types is delivered. When the GET-resource with a type-ID as QueryString is called, the system has to find the corresponding type definition (schema) and send it as response to the client. In case of a request to the GET-resource with an instance-ID as QueryString, the system has to find the matching instance data and send it together with the corresponding type-ID as response to the client. Requests for unknown types or instances are answered with an error.

Behaviour of the POST-Resource

The POST-operation, where the POST-resource is the target of the request, can only be called with a HTTP-POST. The HTTP-GET is not supported and consequently responded with an error. But as HTTP-POST can contain additional data in the request body, two cases have to be differentiated, which are explained next:

POST-Operation with Data

When the POST-resource is requested and data is sent as part of the request body, three different cases can occur: First, if the POST-resource is requested without any type- or instance-ID, a new type has to be created as otherwise a type-ID (type-update) or an instance-ID (instance-update) is required for type or instance identification. Accordingly a new type-ID and information about the created type schema is delivered as response. Second, the POST-resource is called with a type-ID as parameter. This kind of request has two faces: It can indicate that a type has to be updated, but it can also indicate that a new instance has to be created. Which kind of operation is requested can only be decided by taking the content of the request-body into account. If the body contains type information, a type update is demanded and the following response will contain the updated type information. If instance information is sent, a new instance has to be created and the response will contain the created instance data including the new instance identifier. Finally, if the POST-resource is requested with an instance-ID as parameter an update of the defined instance has to occur. As response the updated instance is delivered.

POST -Operation without Data

A request to the POST-resource could also occur with no data sent in the message body. In this case also three different situations could arise: If the POST-resource is called with neither a type- or instance-ID nor any data in the request body, it cannot be decided which action to take. Therefore this kind of request is invalid and answered with an error. A request with a type-ID and an empty request-body must be interpreted as a call for a deletion of a type, as the type is specified and the content of the type shall be empty. Consequently, the response to such a request, a deletion confirmation for that type is sent. And finally, when then POST-resource is request with an instance-ID, this must be understood as a request for the deletion of the specified instance, which is answered by a confirmation, that this instance has been deleted.

Behaviour of the LIST-Resource

In the HTTP-implementation of REST a LIST-operation is realized by using a HTTP-GET with a collection (folder) as URL. In response the client gets a listing of all available resources (representations) in this folder, which can contain all types of representations. A listing by representation type, as far as resources are not sorted by types, is not possible. For the purpose of business data a listing of available types and instances of a certain type is required. Therefore the following functionality is proposed: When the LIST-Resource is called with an HTTP-GET without any QueryString-arguments, the resource generates a complete list of available types as response. The client can then ask for a listing of the available instances of a certain type by calling the LIST-resource with the corresponding type-ID or request the structure of the desired type by using the GET-Resource in combination with the related type-ID. In case of a request for the LIST-resource with a type-ID as QueryString, the resource delivers all available instances of that type as response. The client can then call data of a certain instance by using the GET-resource in combination with the instance-ID of the desired instance. And finally, a call to the LIST-resource with an instance-ID as QueryString would provide the data of the specified instance. But as this functionality is already covered by a call to the GET-resource with an instance-ID as QueryString, this call is redirected to the GET-resource.

Behaviour of the SEARCH-Resource

RESTful applications usually provide a way to search for specific instances. This search "function" is often realized by a special "subfolder" /search/<type> in the URI (e.g. (Mäkeläinen and Alakoski, 2008)). In order to use this functionality, the knowledge of the type schema is required in order to specify which attribute of the type should conform to a specific value during the search. Additionally, this kind of search does not allow a search for available types, as the type has to be defined for a search. FAST ACCESS takes a different approach by differentiating between HTTP-GET and POST requests as well as the transferred values. The different combinations of HTTP-method and passed arguments are explained in the following:

SEARCH-Operation with Get

If the SEARCH-resource is requested with a HTTP-GET without parameters, it has to be interpreted as a demand for available attribute types in order to define the value of a certain attribute in a subsequent POST to the SEARCH-resource. Accordingly, all available basic attributes are returned as response. When the SEARCH-resource is called with a type-ID as parameter, the corresponding type schema is awaited as response. This can then be used to specify a certain value of the schema for the subsequent POST to search for the corresponding instances. And finally, the search for an instance, where instance-ID is sent with the request, would result in one specific instance. Being identical to a request to the GET-resource with an instance-ID as parameter, such a request is redirected to the GET-resource.

SEARCH-Operation with Post

A call to the SEARCH-resource with a HTTP-POST implies that the data transmitted to the framework has to be interpreted as values of the search parameters. As in the other cases three different types of calls to this resource are conceivable: First, when the SEARCH-resource is called with an HTTP-POST and no type- or instance-ID is transmitted, a search for all types, which match with the transmitted data, has to be executed. The response to such a request is a list of types, which fit the search criteria. When a type-ID and data is sent with an HTTP-POST to the SEARCH-resource, the attached data has to be interpreted as search criteria for instances conforming the type-ID. In response to such a request all fitting instances are listed. And, as in the case of a call to the SEARCH-resource with a HTTP-GET and an instance-ID, a POST to this resource is also redirected to the GET-resource.

The presented resources and their respective behaviour cover all necessary CRUD-functions for types as well as instances and, while other behaviours are also conceivable, offer great flexibility for data exchange and integration. Furthermore, the LIST- and SEARCH-resources offer additional functionality, which is regularly needed in business environments. The following section presents the structural concept of the framework, which enables the behaviour of the presented resources.

STRUCTURAL CONCEPT (STRUCTURAL VIEW)

The architectural structure of FAST ACCESS follows a layered architecture model in order to reduce complexity, as each layer has a different view on FAST ACCESS's structural elements, and support uncomplicated modification and extension of certain functions if necessary. As parameters, which are passed between the layers, two standardized, but extensible one-dimensional arrays are used. And if errors occur during request processing, category-based and partly standardized status codes and descriptions are proposed. In the following the proposed parameter arrays, the different layers of the architecture and the status code concept are presented:

Communication between layers

While the layered architectural approach offers many benefits, the communication between the layers is an important point for overall functionality and extensibility. In the proposed concept two one-dimensional arrays, the IncomingMasterArray and the OutgoingMasterArray, are passed between the layers. This approach follows the fundamental structure of HTTP-requests and -responses, where the request and the response consist of a header and a body. In the proposed framework, the mentioned arrays have two rows, containing an array with meta data in the first row and an array with the content data in the second row. The contained data is required by the different layers for correct processing, but all layers are able to handle cases of missing data.

The meta data of the IncomingMasterArray contains a type-ID or instance-ID, if they were part of the request, username and password and, in case of a post, the date of the last update of the transferred content data. Additionally it can be extended, if further data is needed by the different layers. The content part of the IncomingMasterArray contains all data that was sent with the incoming request.

The meta data of the OutgoingMasterArray includes all data that is anticipated to be useful for the requesting client. First of all a status code and description of the response is included in order to allow the client an adequate reaction to different anticipated states of responses. In addition the type-ID respectively the instance-ID of the involved type or instance as well as its creation and last alteration date are part of the meta data. The content part of the OutgoingMasterArray contains all data that is sent as response.

Layers of the architectural concept

As stated above, the structural concept of FAST ACCESS follows a layered approach. These layers include HTTP as transport layer, a request management layer, an authorization layer, a routing layer, a remote connection layer, a local processing layer, an operations management layer and a data management layer. Figure 3 presents the hierarchical concept of the layered approach and each of these layers will be described in the following.

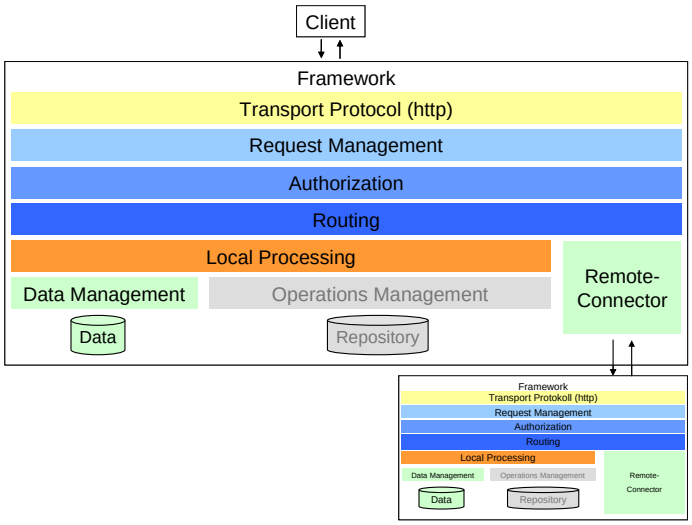

Figure 3: Layered structure of FAST ACCESS

HTTP-Protocol as transport layer

The HTTP-protocol is proposed as basic transport layer. But as modern browsers usually only support the GET- and POST-methods of the HTTP-protocol, only these methods of the protocol are used.

Request Management Layer

The request management layer is responsible for analysing the incoming requests and routes these to the corresponding functions of the lower architectural layers. The layer itself consists of the four presented resources (GET, POST, LIST and SEARCH) and realize the logical "front-end" of the framework. In the first step the request management layer creates the IncomingMasterArray, which is used as parameter transferred between layers. The meta data of the incoming request is copied to a second array and then placed into the first row of the IncomingMasterArray. The content data of the incoming is also copied to an array and is placed into the second row of the IncomingMasterArray. Based on the incoming meta data the respective resource calls the appropriate function of the next layer with the IncomingMasterArray as parameter.

The result of the respective functions is the OutgoingMasterArray. The respective resources render the data of this array to XML and place a reference to the corresponding XSL-resource in order to make a rendering for a human user by a browser possible. The request management layer does not interpret any of the OutgoingMasterArray data.

Authorization Layer

The authorization layer is in charge of validating the authentication and subsequently checking the authorization of the incoming request respectively the outgoing response. Each function of the authorization layer gets the IncomingMasterArray as input from the request management layer, but depending on the requested function the authorization is verified before or/and after calling the function of the next layer. The authentication is based on the username und password, which is passed as part of the meta data of the IncomingMasterArray. The authorization validation is then executed based on the group membership of the user and the requested operation based an access control lists (ACLs) for types and instances. The default authorization strategy is a white list, meaning that access is denied as long as a certain user is not member of the respective group.

As already mentioned above, the authorization has to be checked before or/and after calling the function of the next layer. If a certain instance is requested, the access authorization to this instance can be immediately determined. A list or search request cannot be checked a priori, as the layer does not know which types or instances are delivered as result. Therefore, the authorization inspection has to occur after the response from the next layer. In the case of list or search requests, unauthorized types or instances are deleted from the OutgoingMasterArray and the array is then passed back to the Request Management Layer.

Routing Layer

The routing layer is responsible for directing requests and integrating local and remote data. Based on the requested type or instance of a GET- or POST-operation, the request is routed to the remote connector, if another framework is responsible for the type or instance. If the local framework is in charge for the requested operation, the request is forwarded to the local processing layer. In the case of a LIST- or SEARCH-request, the routing becomes a bit more complex. The request is sent to all, local and remote frameworks (local processing and remote connector layer), which are known for being responsible for the requested list or search criteria. The responses of the involved frameworks are then merged and then passed back to the authorization layer. In the actual version of the framework only one framework, local or remote, can be in charge of a type and the corresponding instances. This is due to the fact that the routing target of a create-instance-operation cannot be resolved if more than one framework is responsible for a certain type.

Remote Connection Layer

The remote connection layer establishes the connection to other known frameworks and therefore realizes the recursive structure of the overall framework concept. The layer has to fulfil the tasks of the preceding layers in a reverse order as it has to consolidate the data for making a valid request to the next framework. Therefore, it first replaces the user identification data (username and password) of the IncomingMasterArray with the username und password of the local framework. This follows the idea that each framework is an entity on its own, as it is standard in a business environment, when a company requests a delivery from its supplier. Subsequent it translates the identifier of the type or instance of the incoming request into the type or instance identifier of the remote framework. Finally, the HTTP-request to the corresponding resource is prepared and executed. The response to GET- and POST-operations are immediately converted into the OutgoingMasterArray and passed back to the routing layer. Responses to SEARCH and LIST-requests are analysed first, as new types or instances at the remote framework could be part of the response. If new types or instances are part of the response, they are registered to local framework as being known types or instances of the corresponding remote framework. This is required as a subsequent request of the original client could ask for one of the previously unknown types or instances. If they were not registered, the routing layer would answer such a request with a type or instance unknown error.

Local Processing Layer

The local processing layer handles all operations, which are determined to be requests to local resources. The layer decides if the basic CRUD-operations can be read/written directly from/to the database or if previous data processing, for example sending an email to third party in case of an instance update, is necessary. This processing can either be a local function or a known web service and therefore, the processing layer can also be seen as connector to regular web service architectures. In order to be able to determine whether additional processing is required, each defined type can have exactly one target function for each CRUD-operation. If no function is defined for a certain operation, the request is transferred to the data management layer for read or write. If a function for prior processing is set, the request is forwarded to the operations management layer.

Operations Management Layer

The operations management layer gets all requests, which require further processing previous to a corresponding response. As this further processing can become very complex, because it could consist of one or more activities, could be processed locally or remote and could be executed by humans or machines (e.g. regular web services), the detailed structure and functionality of the operations management layer is described in an upcoming paper. Nonetheless, even without the operations management layer, the proposed architecture can already be used, when previous processing is not necessary or this processing is executed by the database management system.

Data Management Layer

The data management layer is responsible for reading and storing requested and transmitted data from and to the database. In case of a read, the layer selects the requested type or instance, creates the OutgoingMasterArray and returns it as result of the call. If a write-operation, thus a create, update or delete, is requested, the data management layer first checks if the transmitted data is valid for the given type. If the passed data is valid, it is written to the database, the OutgoingMasterArray is created and returned as result. Otherwise the data management writes an error into the OutgoingMasterArray and returns it to the caller.

As almost all layer layers need to validate, read or write data in order to fulfil their responsibilities, the overall data structure for all layers of this version of FAST ACCESS is presented in Figure 4:

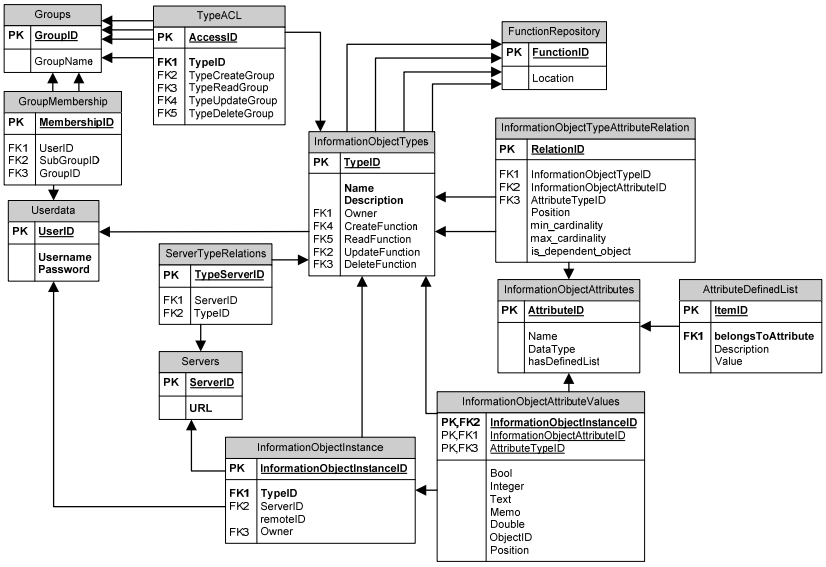

Figure 4: Data Schema of FAST ACCESS

Status Codes

Whenever functions are executed, errors are likely to occur. As in each of the presented layers errors can occur during processing, a standardized list of possible status categories and defined status codes is proposed. These status codes are sent to the requesting client as part of the response header and indicate if an error occurred during request processing. In order to avoid mistakes, which were made during the development of the HTTP-protocol, for each layer a certain category is proposed, which contains certain status codes for errors, which could be anticipated, but also additional codes for possible errors found in the future. This concept allows clients to react in a specific way, if known errors occur, while unknown errors of a certain category can be handled by default actions.

CONCLUSIONS AND FURTHER WORK

The presented system architecture allows the virtual integration, including all CRUD-operations, of data from various sources and has already proven its applicability in many integration projects in university as well as business contexts. Especially in cases where the existing application could not be replaced and had to stay fully functional, FAST ACCESS delivered, due to its unified interface, a comfortable and reliable way to gain access to the underlying data based on atomic transactions. A realization of the same features based on WS-technology would have been much more complex and time-consuming, as for each data entity an own web service would have been necessary. But following Markus, Majchrzak and Gasser (2002): "Only the accumulated weight of empirical evidence will establish the validity" of the proposed system architecture.

When recursively coupled with additional frameworks, the depth is almost unlimited. Additionally, when the data management layer is replaced by specialized connectors, for example for MS Access, MySQL, SQL Server, almost all kinds of data sources, even excel files, can be made available for further use, which is one of the most frequent problems within daily business. The authorization layer, while having to be configured, protects confidential data from unauthorized access.

Nonetheless, it has to be mentioned that data integration based on atomic transactions alone is not applicable in all types of business scenarios. Therefore, additional research is necessary to present ways how the local processing layer and operation management layer have to be structured in order to allow processing before or after data storage or retrieval by the data management layer. This enhancement would also allow the integration of workflow patterns. But if these workflows contain machines and humans as executor of functionality (activity), two other components are necessary: a workflow engine for machine processing and a portal component for human interaction.

REFERENCES

1. Benslimane, D., Dustdar, S. and Sheth, A. (2008) Services Mashups: The New Generation of Web Applications, *IEEE Internet Computing*, 12, 5, 13-15.
2. Berners-Lee, T., Fielding, R. and Frystyk, H. (1996) "Hypertext Transfer Protocol -- HTTP/1.0", http://www.ietf.org/rfc/rfc1945.txt.
3. Bosworth, A. (2004) "ICSOC 2004 keynote talk", http://adambosworth.net/2004/11/18/iscoc04-talk/.
4. Curbera, F., Duftler, M., Khalaf, R. and Lovell, D. (2007) Bite: Workflow Composition for the Web, *In: Conference Proceedings of the International Conference on Service-Oriented Computing – ICSOC 2007*, 94-106.
5. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. (1999) "Hypertext Transfer Protocol -- HTTP/1.1", http://www.w3.org/Protocols/rfc2616/rfc2616.html.
6. Fielding, R. T. (2000) Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine.
7. Hevner, A. R., March, S. T., Park, J. and Ram, S. (2004) Design Science in Information Systems Research, *MIS Quarterly*, 28, 1, 75-105.
8. Mäkeläinen, S. and Alakoski, T. (2008) Fixed-Mobile Hybrid Mashups: Applying the REST Principles to Mobile-Specific Resources, *In: Conference Proceedings of the Conference on Web Information Systems Engineering - WISE 2008*, 172-182.
9. Markus, M. L., Majchrzak, A. and Gasser, L. (2002) A Design Theory For Systems That Support Emergent Knowledge Processes, *MIS Quarterly*, 26, 3, 179-212.
10. Natis, Y. V. (2003) "Service-Oriented Architecture Scenario", http://www.gartner.com/resources/114300/114358/114358.pdf.
11. Papazoglou, M. P., Traverso, P., Dustdar, S., Leymann, F. and Krämer, B. J. (2006) Service-Oriented Computing Research Roadmap, *In: Conference Proceedings of the Dagstuhl Seminar Proceedings 05462, Service Oriented Computing (SOC)*.
12. Parameswaran, M. and Whinston, A. B. (2007) Social Computing: An Overview, *The Communications of the Association for Information Systems*, 19, 762-780.
13. Ropohl, G. (1999) Allgemeine Technologie - Eine Systemtheorie der Technik, Hanser, München.
14. Rosenberg, F., Curbera, F., Duftler, M. J. and Khalaf, R. (2008) Composing RESTful Services and Collaborative Workflows: A Lightweight Approach, *IEEE Internet Computing*, 12, 5, 24-31.
15. Vaishnavi, V. K. and Kuechler, W. (2004) "Design Research in Information Systems", http://www.isworld.org/Researchdesign/drisISworld.htm.
16. Vinoski, S. (2007) REST Eye for the SOA Guy, *IEEE Internet Computing*, 11, 1, 82-84.
17. Weerawarana, S., Curbera, F., Leymann, F., Storey, T. and Ferguson, D. F. (2005) Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More, Prentice Hall PTR.
18. Yu, J., Benatallah, B., Casati, F. and Daniel, F. (2008) Understanding Mashup Development, *IEEE Internet Computing*, 12, 5, 44-52.